ESC-TR-2006-079

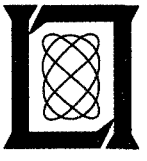
Technical Report
1112

Coverage Maximization Using Dynamic Taint Tracing

T.R. Leek
G.Z. Baker
R.E. Brown
M.A. Zhivich
R.P. Lippmann

28 March 2007

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS



Prepared for the Department of Homeland Security under Air Force Contract FA8721-05-C-0002.

This report is based on studies performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Department of Homeland Security under Air Force Contract FA8721-05-C-0002.

The ESC Public Affairs Office has reviewed this report, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Gary Tutungian
Administrative Contracting Officer
Plans and Programs Directorate
Contracted Support Management

ABSTRACT

We present COMET, a system that automatically assembles a test suite for a C program to improve line coverage, and give initial results for a prototype implementation. COMET works dynamically, running the program under a variety of instrumentations in a feedback loop that adds new inputs to an initial corpus with each iteration. One instrumentation in particular is crucial to the success of this approach: dynamic taint tracing. Inputs are labeled as tainted at the byte level and all read/write pairs in the program are augmented to track the flow of taint between memory objects. This allows COMET to determine from which bytes of which inputs the variables in conditions derive, thereby dramatically narrowing the search over inputs necessary to expose new code. On a test set of 13 example programs, COMET improves upon the level of coverage reached in random testing by an average of 23% relative, takes only about twice the time, and requires a tiny fraction of the number of inputs to do so.

ACKNOWLEDGMENTS

We would like to thank Douglas Maughan at HSARPA for sponsoring this research.

TABLE OF CONTENTS

		Page
	Abstract	iii
	Acknowledgments	v
	List of Illustrations	ix
	List of Tables	ix
1.	INTRODUCTION	1
2.	COMET SYSTEM DESCRIPTION	3
	2.1 Initial inputs	3
	2.2 The coverage frontier	3
	2.3 Dynamic taint tracking and tracing	4
	2.4 The search for new inputs	11
3.	EXPERIMENTS	15
	3.1 Method	15
	3.2 Results	15
4.	SYSTEM TUNING	21
5.	RELATED WORK	23
6.	CONCLUDING REMARKS	27
	References	29

LIST OF ILLUSTRATIONS

Figure No.		Page
1	COMET flowchart	3
2	The coverage frontier	5
3	Taint graph example	6
4	Coverage improvement for the b2 model program	17
5	System tuning	21

LIST OF TABLES

Table No.		Page
1	Condition objective functions.	11
2	COMET performance.	16
3	Sample "hard" conditions from the model programs.	17
4	Conditions in `mutt-utf8` not controlled by COMET.	18

1. INTRODUCTION

Testing remains the primary method by which the conscientious developer assures herself that her code is correct [16]. Test suites for use in regression testing are extremely valuable, so valuable that software companies routinely allow them to grow so large that running them takes days or even weeks [26]. Nevertheless, these suites are typically incomplete in the sense that they cover considerably less than 100% of the lines in the code.

Formal methods such as theorem proving and model checking might seem to offer an alternative, but they continue to encounter strong resistance from the workaday programmer. These methods are tricky to employ and impractical for programs of more than a few thousand lines. Further, it is difficult to keep models and programs synchronized, and so there is a real risk of proving things about models that are not reflected in implementations. Random testing, in which the program is subjected to random draws from some distribution over the inputs, is a simple and more commonly used alternative. In a recent study, this sort of "fuzz testing" was able to crash 73% of the GUI-based applications for OS X [19]. However, while it is clear that random testing can uncover bugs, it is unlikely that random inputs, even those generated with a context-free grammar, will be sufficiently well-formed to exercise the deeper parts of a program.

This paper makes the following new contributions. It presents COMET (COverage Maximization using Taint), a system for automatically obtaining a test suite for a program via a feedback loop that maximizes line coverage directly. COMET works dynamically, running the program under a number of instrumentations, including one that discovers the *coverage frontier* in need of immediate exploration, another that *dynamically traces taint* for memory objects back to individual bytes in the labeled inputs, and yet another that automatically composes objective functions to be used by standard nonlinear optimization techniques to find input byte settings that expose new code. COMET maximizes coverage by adding inputs that uncover new code to a corpus. While it can be initialized with random inputs, one of its great strengths is that COMET can be seeded with a regression test suite. If that suite contains at least some well-formed inputs, then COMET may be able to assemble a test suite that covers vastly more code than if it had started with a single random input. COMET is a working prototype and we have evaluated its performance on 13 illustrative model programs. Performance is compared to a strawman random testing approach and found to be better. Coverage improves by 23% relative, and requires only about twice as much time. System tuning experiments indicate that even better runtimes may be possible with only minimial decrease in final coverage. And COMET uses far fewer inputs than random testing. In our experiments, an average of 30 inputs were equivalent to the ten million used for a random testing baseline.

This paper is organized as follows. In Section 2, we describe COMET in detail, broken down into its three main pieces. Next, Section 3 presents results on some initial experiments. System tuning is discussed in Section 4 and related work in Section 5. Finally, there are some concluding remarks in Section 6.

2. COMET SYSTEM DESCRIPTION

COMET works in a feedback loop, a very simplified version of which is depicted in Figure 1 as a flowchart. The purpose of each of the boxes in the flowchart is described in detail in its own section below. The system is also presented, in more detail, as Algorithm 1. At a high level, in each iteration of the feedback loop, the algorithm considers each of a set of inputs, I_c. It identifies, for each input, a set of conditionals that are both tainted (see Section 2.3) by the inputs and for which only one branch is covered (Section 2.2). For each such conditional, the algorithm searches for an input that exposes the other branch of the conditional (See Section 2.4.2), adding any it discovers to a new set, I_d, for consideration in the next iteration.

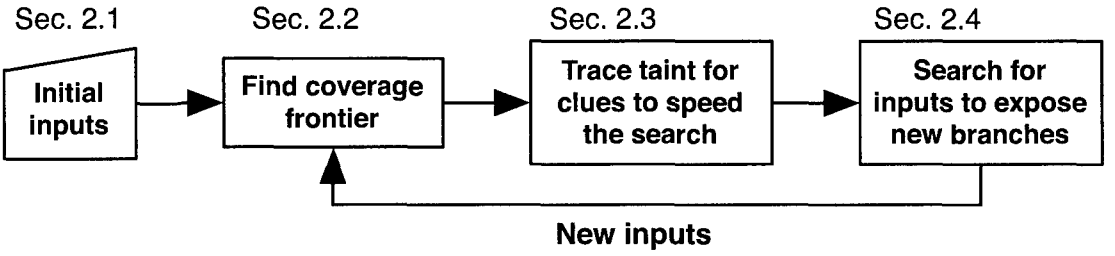


Figure 1. COMET flowchart

2.1 INITIAL INPUTS

For each iteration of the feedback loop, there is some set of inputs, I_c, to be considered. In the first iteration, this set might be filled with draws from some probabilistic grammar, or it might be a regression test suite. This set of *initial inputs* is I_i in Algorithm 1. Subsequent iterations use a set discovered by the algorithm itself, I_d, after discarding overlap with the final corpus of inputs being collected, I_f.

2.2 THE COVERAGE FRONTIER

The first step in COMET's feedback loop is to run the program under a novel instrumentation that determines the *coverage frontier*. This is the set of conditionals for which only one of the two possible branches was ever taken (in Algorithm 1, this is C_{hcc}). This coverage frontier is depicted in Figure 2 as the dashed line separating the covered (black) and not-covered (white) elements of the flowchart for the program. The diamonds in this picture are conditions, one of which is coloured grey to indicate that it is a half-covered conditional (HCC). This conditional lies on the border between covered and not-covered code, and it is where we focus effort. If we can find an input that exposes the other branch of a HCC, we have exposed new code. The HCC instrumentation has two

```
input  : I_i, an initial set of inputs
         P, the program under test
output: I_f, a final set of inputs that gives better coverage than I_i
I_f ← I_i ; I_c ← I_i ;
while |I_c| > 0 do
    I_d ← ∅ ;
    foreach i ∈ I_c do
        C_hcc ← GetHcc(i, P) ;
        TT ← GetTaintTraces(i, C_hcc, P) ;
        foreach tt ∈ TT do
            (s, ni) ← NewInputsSearch(tt, P) ;
            if s = SUCCESS then
                I_d ← I_d ∪ ni ;
            end
        end
    end
    I_c ← I_d − I_f ;
    I_f ← I_f ∪ I_c ;
end
```

Algorithm 1: COMET algorithm.

parts. First, a CIL [20] source-to-source transformation adds code to the start of both the `TRUE` and `FALSE` blocks of every conditional in a C program[1]. The inserted code makes calls to a library of runtime support that maintains a picture of conditional block coverage. The runtime support, written in C, is the second part of the instrumentation, and it includes an API that permits us to clear and inspect the coverage picture on a per-input basis if we so choose, or to collect summary coverage information for a batch of inputs.

2.3 DYNAMIC TAINT TRACKING AND TRACING

The second step in the feedback loop is to determine which of the HCCs involve variables derived from input bytes, and which bytes of which inputs control each variable. In Algorithm 1, this information is represented as TT, which is a set of *taint traces* for the half-covered conditionals C_{hcc}. There will be at least one trace in TT for every variable in a conditional in C_{hcc} that is determined to be derived from input bytes and thus "tainted." If the conditional is in a loop, more than one taint trace may appear in TT (see Section 2.4.3). Each taint trace, $tt \in TT$, indicates the variable traced, the conditional containing it, and the ranges of input bytes from which it was derived. These *tainted half-covered conditionals* in TT are those which guard new code that we may expose by varying inputs. Knowledge of input byte ranges makes the search over inputs to expose new code tractable.

[1]Note that CIL reduces C to a simpler subset of itself, rewriting `switch` statements with nested `if`s, and replacing loops with some combination of `while`, `if`, and `goto` statements. Therefore, this one transformation is adequate to handle all branches.

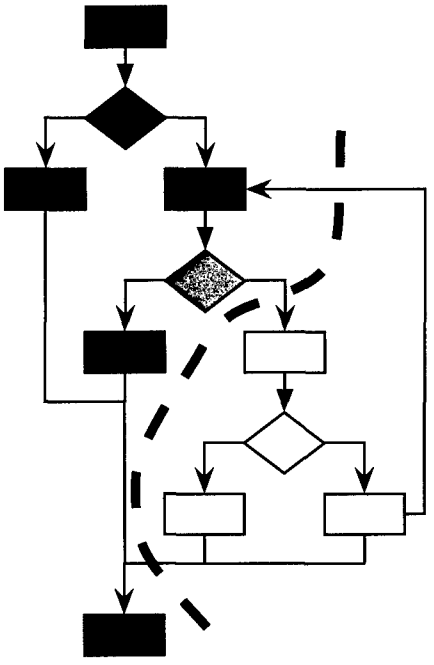


Figure 2. The coverage frontier

2.3.1 Taint Graph Example

A dynamic *taint graph* allows us to determine precisely how conditional variables depend upon inputs. We present a pair of example taint graphs in Figure 3 before going into a more formal description. The graph on the left is the situation before and the graph on the right is the situation after a single byte in a buffer is written with newly tainted data. In the figure, rectangle nodes A and B are buffers in memory. Oval nodes $input_1$ and $input_2$ are sources of taint. These sources are created via calls to a `LabelTaint` function that are currently inserted in the source by hand. Arrows between nodes represent the transfer of taint. The ranges next to an edge give the write (for the node at the arrow head) and read (for the node at the arrow tail) ranges for the transfer. When the ranges are the same, only one is given. In these taint graphs, all edges are *copy edges*, meaning that the write is an exact copy of the read (see Section 2.3.2). Thus, in the left-hand graph 3(a), the first five bytes of B are a tainted, exact copy of bytes 5..9 of A. And bytes 5..9 of A are a tainted, exact copy of bytes 5..9 of source $input_1$. In the right-hand graph 3(b), byte 0 of source $input_2$ has been written to byte 7 of A. We update the graph in two steps to reflect this fact. First, we delete taint for byte 7 of A. In order to preserve as much information as possible, we first split in two the taint edges from $input_1$ to A and from A to B, as the 7th byte of $input_1$ no longer taints A. Additionally, in order to properly maintain the taint relationship between the 7th byte of $input_1$ and the 2nd byte of B, a new edge is introduced directly from $input_1$ to B. The second step is simply to add a new edge from $input_2$ to A.

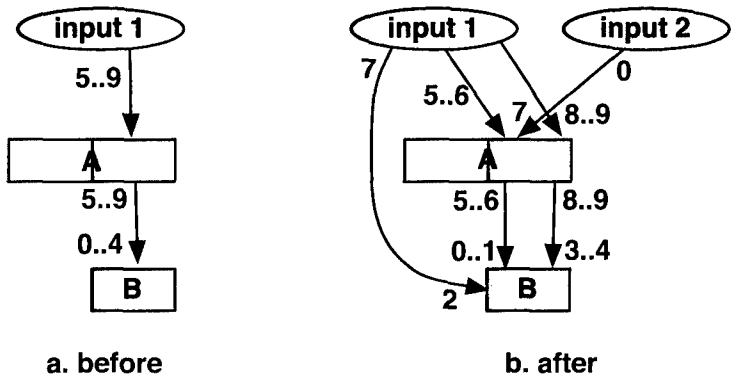


Figure 3. Taint graph example

2.3.2 The Dynamic Taint Graph

Nodes in a graph represent objects on the heap or stack, such as buffers and variables. Directed edges represent the transfer of tainted data between objects. The tail of an arrow is labeled with the range of bytes read, and the head with the range of bytes written. There are copy and compute edges.

1. A *copy edge* corresponds to a direct copy of data, with the size of the byte range read and written being the same. Both simple assignments (e.g., `a=b`) and a few common standard library calls that copy bytes exactly create this sort of edge.

2. A *compute edge* corresponds to any other read/write pair. The size of the byte ranges read and written need not be the same; all we know is that the write range is derived from the read range. Complicated assignments (e.g., `a=b+c+a`) and some standard library calls result in this sort of edge.

The reason for keeping distinct the two types of edges is that when it comes time to trace taint back along a graph edge, we will be able to do so much more precisely for a copy than for a compute edge. Say `p` is a pointer to the first byte of B in Figure 3(a). If we are asked to trace taint for this byte, we can do so very precisely in two steps: byte 0 of B is a direct copy of byte 5 of A; byte 5 of A is a direct copy of byte 5 of $input_1$. Conversely, if the edge from A to B had been a compute edge, the trace widens in the first step because the best we can say is that byte 0 of B derives from bytes 5..9 of A.

Instrumentation is again via CIL, and so the range of syntactic constructs with which we must contend is smaller than it would be for the original C source. There are only three places in a CIL program in which taint may transfer and we instrument all three:

1. Assignment statements

2. User functions, via shadow call and return stacks

3. Library functions, via models

Maintaining the taint graph requires just two runtime support functions. The first of these is `AddTaintDependency(p1,n1,p2,n2,c)` which simply adds an edge to the graph between the two nodes representing the objects pointed to by $p1$ and $p2$, labeling it with read range $[p1, n1)$ (by which we mean bytes $p1..p1 + n1 - 1$) and write range $[p2, n2)$. The parameter c is a boolean set to *true* iff the edge being added is a copy edge. This function is called to extend the graph by introducing a new taint dependency.

The second function, `DeleteTaint(p,n)`, deletes all taint from the graph on the byte range $[p, n)$. It is called when a previously tainted range of memory is overwritten and preserves taint that previously flowed through these bytes to other memory objects. Deletion (alternately *untainting*) is a complicated operation on the taint graph as it may necessitate re-routing edges to preserve data-flow information. It is described briefly in Algorithm 2. Taint deletion makes use of a number of functions. $PtsTo(p)$ resolves a pointer p back to a node in the taint graph (see Section 2.3.7). $InEdges(o)$ and $OutEdges(o)$ return the set of incoming and outgoing edges for node o in the graph. $WriteRange(e)$ and $ReadRange(e)$ return the write and read byte ranges, respectively, for edge e. The first and second steps in Algorithm 2 add new edges to represent the portions of incoming and outgoing edges that *do not overlap* the deletion range. Any such edge is also added to set DS earmarked for later deletion. Thus, for Figure 3(b), we add four new edges. We add two copy edges from $input_1$ to A with ranges 5..6 and 8..9. And we add two new copy edges from A to B: one from 5..6 to 0..1 and another from 8..9 to 3..4. Further, we add two edges to DS: the two original edges connecting $input_1$, A, and B. The third step re-routes the portion of any incoming/outgoing edge pair that overlaps the deletion range. In Figure 3(b), this adds an edge from byte 7 of $input_1$ to byte 2 of B. The fourth step deletes all the edges in DS, which are no longer needed. This means, in Figure 3(b), that the edge for range 5..9 between $input_1$ and A goes away, as does the edge for 5..9 to 0..5 between A and B. Note that considerable complexity has been elided away here: for instance, re-routing edges has to proceed differently for copy and compute edges.

2.3.3 Instrumenting Assignments

We use *lhs* to stand for the lvalue that is the left-hand side of an assignment, and *rhs* for the expression that is the right-hand side. If we assume a function that recursively examines *rhs* and assembles a set of lvalues, $lvalues_{rhs}$, discovered therein, then we get Algorithm 3 for instrumenting an assignment to transfer taint. Here, the action of `AddCall(fn,args)` is to insert a call to `fn` with parameters `args` into the source code. Calls are inserted in sequence and immediately prior to the current assignment under inspection. This means that an assignment will have the following dynamic effect upon the taint graph. First, taint will be deleted for the node representing the object that is the *lhs*. Second, for each lvalue contained within *rhs*, a new taint edge representing the influence of that lvalue upon *lhs* is added to the graph. It might appear that assignments like `a=a+b`, in which *lhs* appears as part of *rhs*, demand special treatment; the algorithm would add instrumentation in this case that erases any dependency of variable `a` upon its ancestors (incoming

$DS \leftarrow \emptyset$;
/*Step 1: Add edges to represent non-overlapping parts of in-edges. */
foreach $ie \in$ *InEdges*(*PtsTo*(p)) **do**
  if *WriteRange*(ie) $\cap$ $[p, n) \neq \emptyset$ **then**
    Add taint edge(s) that excludes $[p, n)$ on write range of ie. ;
    $DS \leftarrow DS \cup ie$;
  end
end
/*Step 2: Add edges to represent non-overlapping parts of out-edges (omitted). */
/*Step 3: Reroute in-out edge pairs that overlap the deletion range. */
foreach $ie \in$ *InEdges*(*PtsTo*(p)) **do**
  foreach $oe \in$ *OutEdges*(*PtsTo*(p)) **do**
    if *WriteRange*(ie) $\cap$ *ReadRange*(oe) $\cap$ $[p, n) \neq \emptyset$ **then**
      Add edge(s) to re-route intersection between write, read, and deletion ranges. ;
    end
  end
end
/*Step 4: Delete edges that overlapped $[p, n)$. */
foreach $e \in DS$ **do**
  `DeleteTaintEdge`(e)
end

Algorithm 2: *DeleteTaint*(p, n).

edges in the taint graph) in this case, which is incorrect. To ensure that this is handled correctly, we use a simple transformation to divide this sort of assignment in two by way of a temporary variable, thus `a=a+b` becomes `tmp=a+b; a=tmp`. The function *IsCopy*(*rhs*) determines statically if an assignment is a copy and thus if the new taint edge is a copy edge. This will be the case if *rhs* is a single lvalue that has not been cast to change its bit width. Note that CIL makes all implicit C casts explicit. Referring back to Figure 3, the second graph (b) results from a single assignment, `A[7]=input2[0]`, which is instrumented as two calls to the taint graph runtime system:

```
DeleteTaint(&(A[7]),1);
AddTaintDependency(&(input2[0]),1,&(A[7]),1,TRUE);
```

`AddCall` (DeleteTaint, &lhs, sizeof(lhs)) ;
$c \leftarrow$ `IsCopy`(*rhs*) ;
foreach $r \in lvalues_{rhs}$ **do**
  `AddCall` (AddTaintDependency, &r, sizeof(r), &lhs, sizeof(lhs), c) ;
end

Algorithm 3: Inserting instrumentation for an assignment statement.

2.3.4 Instrumenting User Functions

We instrument functions for which we have source code to be able to transfer taint both into and out of them. Because transferring taint into and out of a function is really just cross-scope assignment, it is covered by Algorithm 3. Allow rhs_i to refer to the ith actual (function argument), and lhs_i to correspond to the formals in the function definition. Upon entering a function, the taint graph needs to be updated as if the assignments $lhs_i = rhs_i$ have just taken place. The only difficulty is that we cannot constitute calls to *DeleteTaint* and *AddTaintDependency* directly, as we don't have the addresses of $lvalues_{rhs_i}$ and lhs_i together in one place. To bring these values together, we employ a shadow stack. At the call site, for each rhs_i, we add instrumentation to push the address, size, and position of each lvalue in $lvalues_{rhs_i}$ to a stack. Likewise, at the head of the function itself, we arrange to pop these pairs off the stack, associate them with formals, and execute the appropriate calls to *DeleteTaint* and *AddTaintDependency*. Similarly, and more simply, transferring taint out of a function is another cross-scope assignment in which taint transfers from the return value expression rhs to the lhs of the function call itself, if any. Our implementation does not currently handle variable length argument lists: the taint graph is only updated for those formals explicitly listed in the function definition. We are investigating detecting and instrumenting common `va_list` idioms.

2.3.5 Instrumenting Library Functions

Functions for which we do not have source code are handled, incompletely, via models of taint transfer. We use a little language to specify how taint transfers across a few very common standard library functions: `memcpy`, `memmove`, `bcopy`, `strcpy`, `strncpy`, `strlen`, and `sprintf`. When CIL encounters one of these calls, it interpolates the actuals with the model for that library function and inserts the correct call(s) to the runtime support to transfer taint across that function. For instance, `memcpy` is augmented as follows:

```
DeleteTaint(dst,len);
AddTaintDependency(src,len,dst,len,TRUE);
memcpy(dst,src,len)
```

2.3.6 Taint Tracing

Input byte ranges for variables in conditionals are found by a depth-first search, augmented to make use of write and read byte ranges on edges and to know the difference between copy and compute edges. This search or *taint trace* is sketched in Algorithm 4, which works by examining the incoming edges for the node corresponding to the byte range being traced, and then recursively following the portion of each that overlaps the trace range. Three new functions make an appearance here. $SrcNode(ie)$ returns the node corresponding to the source of directed edge ie (the parent or node at the tail of the arrow), and $IsTaintSource(o)$ returns true iff node o was introduced by a call to *LabelTaint*. $TranslateRange(i, ie)$ determines the offset of byte range i with respect to the write range of edge ie and then translates that into a byte range with respect to the read range of ie. Notice that this function must do its work differently for copy and compute edges, in the first case carefully preserving the magnitude of the taint range when translating it onto the read range,

and in the second case widening the trace to correspond to the entire read range. Consider, as an example, how the algorithm would trace taint for byte $B[1]$ in Figure 3(b). The original call would be `TraceTaint(&(B[1]),1)`. Only the incoming edge from A to B with write range 0..1 overlaps this trace range. Translating the overlap back to the read range results in a second call tracing taint for byte $A[6]$: `TraceTaint(&(A[6]),1)`. In this case, only the incoming edge from $input_1$ to A for range 5..6 overlaps the taint range. The source node or parent for this edge, $input_1$, is a taint source, and so recursion terminates and we emit a message indicating that the trace leads to byte 6 of $input_1$. Such a precise trace, indicating that a single input byte controls a single byte in a buffer, is only possible because all of the edges traversed in the search are copy edges. Note that the tracing algorithm as described is vulnerable to cycles. To avoid this difficulty, we memoize the function `TraceTaint` and halt recursion if called twice with the same parameters.

```
foreach ie ∈ InEdges(PtsTo(p)) do
    i ← WriteRange(ie) ∩ [p, n) ;
    if i ≠ ∅ then
        p1, n1 ← TranslateRange(i, ie) ;
        if IsTaintSource(SrcNode(ie)) then
            Emit trace result: [p1, n1)
        else
            TraceTaint (p1,n1)
        end
    end
end
```

Algorithm 4: TraceTaint(p,n)

2.3.7 Implementation and Runtime Issues

The taint graph instrumentation is a 2K line Ocaml module extending CIL and is supported by 5K lines of runtime code written in C. The ability to dynamically resolve a pointer to its referent object (the *PtsTo* function assumed above) comes from Ruwase's CRED bounds-checking extension to gcc [27] (itself a modified version of Jones and Kelly's work [13]). CRED objects are bundled with a number of useful pieces of information, including base address, extent, element size, and creation location. CRED inserts code into a program to create and delete these objects as well as providing efficient access to them via a splay tree. The runtime system periodically examines the taint graph, looking for opportunities to compact it. For example, copy edges that are adjacent and compute edges that subsume one another are combined, and islands are removed. In future, we may pursue more aggressive compression strategies, such as reclaiming interior nodes corresponding to temporary variables introduced by CIL. We will never need to trace taint for these variables.

2.4 THE SEARCH FOR NEW INPUTS

The third and final step in the COMET feedback loop is to search for inputs that expose new code. That search is made much easier by the taint graph, as well as by a final new instrumentation, *condition objective functions* that generate a signal indicating how close we are to exposing a branch of a HCC for a given input. The search itself is a combination of heuristics and standard nonlinear optimization.

2.4.1 Condition Objective Functions

The *condition objective function* instrumentation adds two print statements immediately prior to every conditional, one for each branch. These functions are automatically composed (again, they are inserted by CIL) to minimize to zero (after clipping to $0..\infty$) when the conditional exposes the desired branch. A sampling of these objective functions for some common conditions appears in Table 1. These functions have much in common with those developed earlier, by others [17], even though they were separately derived for this work. CIL replaces conditions involving the logical connectives `&&` and `||` with control flow (sequence and `gotos`), so there is no need to include them in the calculus of objective functions. Thus, if we can by some automated search discover a way to set the bytes in the input indicated by the taint trace to the right values, we minimize the condition objective function and expose new code for future exploration.

cond	$obj_{true}(cond)$
`(a == b)`	$(a-b)^2$
`(a != b)`	$1-(a-b)^2$
`(a > b)`	$(b-a+1)$
`(a <= b)`	$(a-b)$
`a`	$obj(a \neq 0)$

TABLE 1. Condition objective functions.

2.4.2 Search Overview

The search for byte settings that will minimize a condition objective function happens, at a high level, in two stages.

1. First, we try some reasonable guesses from two very different sources.

   (a) **Static hints:** If the condition is something like `(c==32)`, then a setting of 32 is an excellent first guess. Likewise, if the condition is `(c<32)`, then 31 would be a good guess. We scan the condition for all integer literals and add each to the list of guesses to try.

(b) **Dynamic hints:** Another (very simple, and therefore not described above) instrumentation inserts print statements prior to every conditional to output the dynamic value of every unique lvalue involved. COMET examines the logfile produced by the instrumented program for these *dynamically determined lvalues*, which it adds to the list of guesses to try.

Once there is a list of guesses, COMET iterates over them and tries each, as well as its successor and predecessor (± 1), checking the condition objective function after each try to see if the guess was correct.

2. Second, we resort to nonlinear optimization, in which the bytes are first considered as a single value in both big and little endian encoding, and next as a vector of bytes parameterizing the objective function. In the first case, the optimization is in one dimension and so we use a golden section search [25]. In the second, it can be in two or more dimensions, and we use the downhill simplex method of Nelder-Mead [21]. Both search techniques were chosen for ease of implementation.

2.4.3 Search Refinements

A variety of details were omitted from Algorithm 1 and will be presented here. Some aim at speeding up COMET while others are expected to improve coverage.

1. **Loops and utility functions:** If conditional c is in a loop or a utility function for which there is more than one call site, then COMET will be confronted with a temporal *sequence of taint traces*, each providing different clues as to which bytes in which inputs control the conditional. Some branches cannot be exposed unless we pay attention to the second, or perhaps later, taint traces. We number the traces as we encounter them in the taint log file with an increasing integer, t, and introduce a probabilistic element to Algorithm 1, whereby the larger is t the lower the probability that the algorithm will use that taint trace to try to expose new code. More precisely, we compute an *initial flip probability*, p_{rf}, for a conditional as an exponential function of t, further shaped by a parameter β.

$$p_{rf}(c) = e^{-\beta t} \tag{1}$$

If $0 < \beta \ll 1$, then the initial flip probability will be close to one, regardless of t, meaning that we essentially use all taint traces, individually, to try to expose new code. On the other hand, if $\beta \gg 1$, then the algorithm mostly pays attention to the first few traces. It should be noted that static solutions to these problems exist: we might unroll loops and make a new uniquely named copy of a function for each call site. Our solution was vastly simpler to implement and handles difficult cases introduced by more esoteric control-flow like `gotos`, recursion, and function pointers.

2. **Re-flip probability:** In some cases, we need to be willing to consider the same conditional over and over, as that will be what is required to achieve the correct state to be able to uncover

new code. Thus, whenever the algorithm consults the threshold p_{rf} to determine whether or not to consider a conditional, p_{rf} is decayed by multiplying it by another parameter $0 < \alpha < 1$, so that reconsideration becomes less likely the more it happens.

3. EXPERIMENTS

3.1 METHOD

We tested COMET against a set of 13 representative example programs of a few hundred lines of code each[2]. Ten of the thirteen were models of real vulnerabilities developed for an evaluation of static and dynamic buffer overflow detectors, and we included them here because they represent real code with known security problems. The other three were included for comparison with prior work. The initial corpus of inputs was taken to be 10 random inputs, of which we retained only those that cause a change in coverage. For the models, we additionally had one well-formed test case each from previous evaluations, and so in this case the initial corpus included that input as well.

We compare performance with the simplest alternative technique: random testing without a grammar, meaning a large number (10 million) of strings drawn from a uniform distribution, taking the length of the string to be the same as for the well-formed input (or some reasonable guess when no such input existed).

A few further notes on these experiments are in order. First, it was necessary to add code to all of these programs in order to conform to the test harness, enable labeling of input as tainted for the taint graph to work, and to efficiently generate random strings inside the program for random testing. This extraneous code amounted to 20–50 lines per program, and we have carefully excluded these lines from all calculation of percent coverage and LOC. Also, it should be noted that we added the source for the function `dn_expand` to the model programs b1 and b2, that of `dn_skipname` to b3, and replaced the function `isdigit` with a macro in `pintos-printf`. These very minor modifications were necessary in order for the taint graph to be able to trace taint through these functions. Discovering these deficiencies was trivial as the taint graph runtime complains if it encounters a function into which it cannot transfer taint.

These experiments were performed on a single Dell Precision 650 workstation with dual 2.66GHz Intel Xeon processors, 2 GB of RAM, and running Linux kernel 2.4.20-8. COMET was made to timeout if it ran for more than 10 minutes to expedite initial experimentation. We obtained these results by setting the free parameters of COMET to $\alpha = 0.9$ and $\beta = 0.01$.

3.2 RESULTS

The results for all 13 example programs appear together in Table 2. The first column of this table is the name of the program under test, and the four sections into which the rows of the table are divided correspond to the four sub-subsections of detailed results and discussion presented below. The second column gives the final percent of lines covered by the entire corpus of inputs assembled by COMET. The third column gives the ratio of that final coverage to what is obtained with random testing. Thus, for program b1, COMET reveals 65% *more* lines than random testing.

[2]Lines of code or LOC, as reported everywhere in this paper, was measured by the `gcov` coverage measurement tool. These numbers are generally about half what would be reported by SLOCCOUNT [33]

	Percent coverage		Time (sec)			
Program	COMET	Ratio wrt RT	COMET	Ratio wrt RT	Num inputs	LOC
b1	87.46	1.65	384	1.10	57	295
b2	85.19	1.82	396	0.92	54	378
b3	86.11	1.51	13	0.35	6	72
b4	97.94	1.28	41	0.27	4	97
s1	92.46	1.16	553	2.10	41	199
s3	87.10	1.00	139	1.38	7	62
s4	76.29	0.85	605	4.41	8	97
s5	90.67	0.99	601	10.36	16	150
s6	100.0	1.17	20	0.13	6	61
s7	72.16	1.36	9	0.65	10	194
mutt-utf8	94.37	0.97	24	0.69	16	71
printf	100.00	1.00	12	0.46	181	97
triangle	100.00	1.20	8	0.25	7	54
mean	89.98	1.23	215.8	1.78	30.8	140.5
median	90.67	1.17	41	0.69	10	97

TABLE 2. COMET performance.

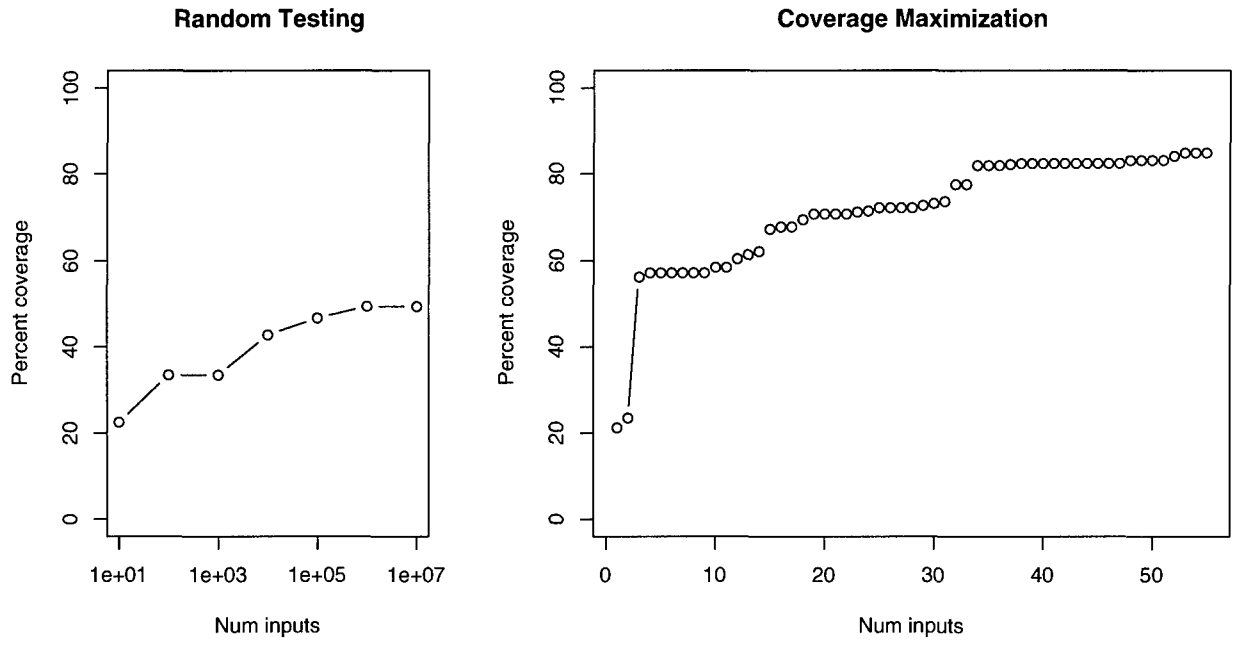
The fourth column gives the time taken by COMET, and the fifth gives the ratio to the time taken by random testing. The sixth column is the final number of inputs in the corpus. Lastly, the seventh column gives the size of each example program in LOC. The last rows of this table present the mean and median for each column.[3]

3.2.1 Model Programs

The ten model programs (b1–b4, s1, s3–s7) are excerpts from historical versions of bind and sendmail known to be vulnerable to buffer overflows. They were created for a previous evaluation of static analysis buffer overflow detectors [37,38]. These excerpts were carefully crafted to contain the overflow plus as as much attendant control-flow complexity as possible, without including so much code that static analysis tools would be unable to analyze them. Thus, these samples of real open-source code are of considerable interest to the security-minded, as they contain examples of the sort of constructions and errors that are symptomatic of real vulnerabilities.

The performance of COMET with respect to random testing on the model programs varied dramatically, ranging from an improvement of 82% for b2, to a degradation of 15% for s4. Figure 4 is

[3]We were only able to test mutt-utf8 with 10,000 examples, which took 0.035 seconds. If we tried many more than that, the program would crash, no matter what random seed we chose. If we extrapolate to how long it might take to actually test with 10M inputs, we get 35 seconds, which makes the time ratio for this program 0.69, the number reported here.



(a) Random testing

(b) COMET

Figure 4. Coverage improvement for the b2 model program

two juxtaposed plots contrasting the temporal progress of random testing with that of COMET for the b2 model. On the left is random testing, which requires a log scale in the x-axis to accommodate the 10 million inputs required to obtain a final coverage level of 46.83% in 429 seconds. On the right is COMET, which attains a coverage level of 85.19% with only 54 inputs, after working for 396 seconds. COMET, in this case, takes a little less time than random testing to achieve its best result. However, it exposes far more LOC and requires much fewer inputs, effectively doing test-set reduction inline with coverage maximization.

The models for which COMET appears to fail are s3, s4, and s5, and Table 3 gives a few examples of the sort of conditions that the system cannot control. For s3, eight lines are not covered, of which five are obviously dead. The other three lines involve variables that are tainted by implicit

model	condition
s3	`(++nchar > maxlen)`
s4	`(fbufp >= &fbuf[MAXLEN])`
s5	`(cmncnt <= 0 && state != QST)`

TABLE 3. Sample "hard" conditions from the model programs.

loc (line)	block size	condition
68	1	`(n>1 && !(ch >> (n*5+1)))`
81	1	`(ch & ~0xffff)`
105	2	`(u8len)`

TABLE 4. Conditions in `mutt-utf8` *not controlled by COMET.*

information flow (IIF) and are therefore opaque to our procedure [28]. One such variable is the counter `nchar` in the first row in Table 3, the value of which is determined by the contents of the tainted buffer being parsed, but not in a data-flow sense. For s4, 23 lines remain uncovered. Three are dead and 19 are guarded by variables derived from inputs via IIF. One IIF-tainted condition, which involves a pointer `fbufp` whose value is determined by the contents of a buffer controlling a parsing loop, appears in the second line in Table 3. The one remaining condition in s4 can, in fact, be controlled by COMET, if the optimization loop is allowed to run for more than the allotted 10 minutes. For s5, 14 lines remain to be covered. One of these lines is dead and one is, again, tainted by IIF as in s3 and s4. The remaining 12 lines are guarded by a more complex IIF-tainted condition involving a state variable that changes according to a state-transition matrix which is indexed by the current character being parsed. An example of this sort of condition appears in row three of Table 3. To summarize, of the 45 lines remaining uncovered in s3–s5, fully 80% are guarded by conditions that are tainted in an IIF rather than a dataflow sense.

3.2.2 Mutt's UTF8 Routine

Mutt is a popular email client whose UTF8 to UTF7 conversion routine was found to have a buffer overflow in version 1.4 by Securiteam in 2003 [29]. The final coverage for this program achieved by COMET is 94.37% of the lines. COMET misses four lines, none of which appear to correspond to dead code. These lines are guarded by the three conditions reproduced in Table 4. The first two involve an integer `ch` that is built up from successive input bytes contained in the buffer `u8` via this snippet of code:

```
ch = (ch << 6) | (u8[i] & 0x3f).
```

Correctly controlling a conditional dependent upon this value would at least require reformulating the taint graph to work at the bit level. The third conditional `(u8len)` appears immediately after a loop that parses the tainted input. Its business is to test if any of the bytes of that input remain unconsumed after parsing is complete (`u8len` is decremented whenever a byte is consumed), thus indicating that the input was ill-formed. The value `u8len` is thus tainted in a rather complicated IIF sense that embodies the purpose of the whole parsing loop. It is worth noting that random testing misses only three lines for this example, exposing the other branch of one of the first two conditions by chance.

3.2.3 Pintos' Version of printf

Pintos is a tiny operating system used to teach college students about operating systems. Its `stdio` implementation contains a simple version of `printf` missing many features of the full-blown `glibc` version. We were unable to harness `printf` directly, as it involves building up a string using the contents of the second, third, and so on arguments, i.e., the `va_list`. COMET would attempt to process, for instance, a format string such as `"%20s"` but would fail when it found no matching string argument. Instead, we arranged to test the format-processing function `parse_conversion` directly. In truth, this also uses `va_list`, for something like `printf("%.*s", max, s)`, which must obtain the value of integer `max` to know what width to make string `s`. We worked around this difficulty by generating a random integer as this width and proceeding. In this final form, COMET covered 100% of the lines in `parse_conversion` in 12 seconds and came up with 121 test cases. For this test, we took the length of the format string to be 5 characters. It should be noted that random testing as easily covered every line in this code, taking 26 seconds and using 10 million examples. Once again, our result is on par with random testing, but our test cases are far fewer and therefore each is more diagnostic.

3.2.4 The Triangle Program

We also tried COMET on the oft-cited triangle program, in which three integers are input and subsequently it is determined whether or not they might make a valid triangle if each were considered a length of a side [14, 18]. The system achieves 100% coverage in 8 seconds, and comes up with 7 inputs. By contrast, random testing (the three sides are random signed integers) makes no progress in revealing the two lines corresponding to identification of isosceles and equilateral triangles even after 10 million guesses. This is apparently considered a difficult program for coverage improvement systems, as only a very small portion of the input space corresponds to these special triangles. Why, then, does this program present no challenge to COMET? Consider, for instance, a condition like `(a == b)`, used to determine if two sides have the same length. Dynamic hints (see Section 2.4.2) tell us the values of both `a` and `b`. Further, the taint graph tells us precisely which bytes of which inputs determine both values, regardless of the encoding used (we encoded the three integers as 12 continguous bytes in a file). No search is required; we simply set the value of `a` to the dynamic value of `b` and the conditional is made true. Other hurdles in the triangle program fall as readily.

4. SYSTEM TUNING

COMET has two parameters, α and β, and two features that can be enabled (or not) at the command line. We did some experiments to explore the trade-off between coverage and runtime resulting from various combinations of parameters and enabled features. The results are presented in Figure 5.

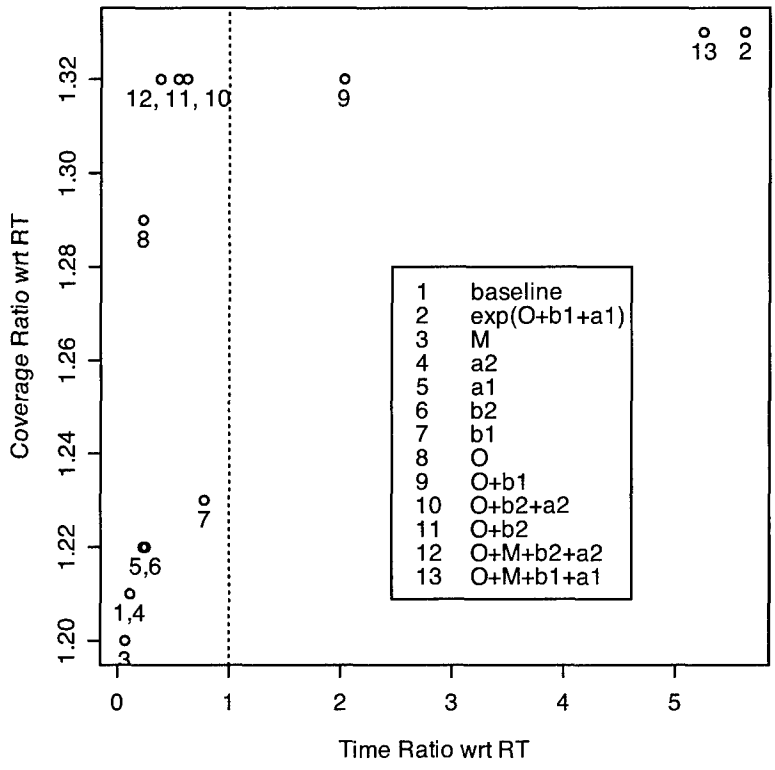


Figure 5. System tuning

It should be said that system tuning happened well after the experiments discussed in Section 3; the system used there corresponds to the one with the best coverage and worst runtime in the figure, i.e., "exp(O+b1+a1)". Also, it should be said that we did this tuning only on a subset of the model programs: b1–b4, s1, s3, s5 and s6. The vertical axis in the plot is the mean (over the eight programs in the subset) coverage ratio with respect to random testing. The horizontal axis is the mean runtime ratio with respect to random testing, and thus any point on the dotted vertical line running through 1 would represent COMET settings with the same runtime as random testing. The rows in the legend of this plot are numbered and represent the different configurations of the system tested. Those numbers appear next to the points on the plot as labels to indicate the coverage and time ratios for each configuration. "a1" corresponds to $\alpha = 0.9$ and "a2" to $\alpha = 0.25$. "b1" and "b2" correspond to $\beta = 0.01$ and $\beta = 0.2$. "O" means that nonlinear optimization is

enabled. "M" is a simple memoization scheme that remembers which blocks have been exposed and treats any branch COMET exposes as a side effect as if it were exposed purposely. Thus, the point labeled 9 corresponds to a configuration of "O+b1", meaning that nonlinear optimization is on and $\beta = 0.01$. The "baseline" system has all features effectively disabled: no optimization, no memoization, $\alpha = 0$, and $\beta = 100$. Regardless, it achieves coverage that is 21% better than for random testing and takes only about one-tenth the time. The biggest gain in coverage is for turning on nonlinear optimization (point "8:O"). Settings for α and β that deviate from baseline (points "5:a1" and "7:b1") offer smaller improvements. This plot indicates that the setting that gets the best coverage (1.33 for point "2:exp(O+b1+a1)") comes at a cost of 5.62 times the runtime of random test. Conversely, the setting for which we get the best runtime ratio (0.07 for point "3:M") produces the lowest coverage. A very attractive operating point appears to be "12:O+M+b2+a2" setting, with a coverage ratio of 1.32 and runtime ratio of 0.38. This configuration gets coverage very close to the best system and is almost three times faster than random testing.

5. RELATED WORK

Previous researchers have used static and dynamic approaches to generating test cases, both individually and in combination. Static approaches typically involve some variation on the theme of examining the source code and collecting constraints along paths and then using a solver to obtain inputs that satisfy the constraints [2,10]. In general, these static efforts are hobbled by a variety of approximations required to render their analyses tractable. For instance, it is impossible to know statically with any accuracy what every pointer in a complex C program refers to [12]. Further, it is difficult to reason about arrays, traces are conflated into paths, and fixpoint solutions are often used for loops. Thus, for a large program, the constraints on the input we can reasonably derive can be too vague to be of much use. Static approaches to test case generation have mainly been applied to very small programs lacking complicated and looping control flow.

By contrast, random and dynamic testing involve actually running the program, using its responses to determine the set of sentences it accepts. Random testing has been used to test the robustness of programs, both in a simple way, subjecting a program to random strings of input [19], and more elegantly by taking draws from a grammar [15]. However, these techniques are of limited use when complex, structured inputs of any length are required. Without considerable prior knowledge and engineering of input generators, it is extremely unlikely that random inputs will expose much past input validation and error-handling code.

Dynamic testing is the approach that most directly inspired this work. "Automated Test Data Generation" or ATDG and its descendants employ a variety of hill-climbing techniques to uncover new code by varying inputs [7, 8, 17, 18]. Success is measured in terms of the fraction of code exposed and the runtime required to do so. These measures are clearly coupled. Given unlimited runtime, we can enumerate every possible input (up to a given length) and are likely to cover much of the code. However, runtime is limited in practice and so ATDG innovation focuses on guiding the search through instrumentation and ordering it via the choice of search strategy. Thus far, the most sophisticated instrumentation employed appears to have been condition objective functions. The best attempts at ordering the search to make it proceed faster seem to have been using a genetic algorithm (GA) and a static dependency analysis that tries to expose code that is likely to be required in order to expose other code ("chaining"). COMET uses four distinct instrumentations: half-covered conditionals to tell it what part of the program to focus on next, the taint graph to tell it from what bytes of what inputs the variables in a conditional derive, dynamically determined lvalues for good hints about what to set those input bytes to, and finally condition objective functions to guide nonlinear optimization. In truth, most of the time COMET avoids search entirely, as the taint graph plus static/dynamic hints tell it exactly how to change the input to expose the code in question. For the model programs, we found that 76.6% of the time static and dynamic hints sufficed to control a conditional. Further, for fully half the remaining conditionals, nonlinear optimization succeeded in exposing the other branch. Note that ATDG requires knowledge of how the inputs are coded, e.g., in order to choose a GA representation for which mutation and crossover operations will usefully explore the input space. For COMET to work, we need only label the input bytes; the taint graph dynamically and precisely reveals the relationship between any variables we query and the input.

There has been considerable convergence between static and dynamic approaches in recent years, so that combination approaches are beginning to constitute a new category. For example, the DSD-crasher system employs dynamic invariant detection, followed by a static analysis that uses a SAT-solver to find inputs likely to break each invariant, finally ending in a dynamic instantiation to verify that the inputs truly violate the invariant [6]. The SYNERGY system (as well as other related efforts [36]) combines testing and verification, allowing each to guide the other in a tightly coupled loop, searching for a test case that reveals a bug whilst simultaneously seeking a proof of its nonexistence [11]. However, SYNERGY currently only works for single-procedure C programs with integer variables.

Directed testing systems like CUTE, DART, and EXE also fall into this combination category [3, 4, 9, 30]. EXE (previously called EGT) uses a mixed symbolic-concrete evaluation of the program that attempts a bounded exploration of all possible traces, "forking" the analysis at each branch point. It fixes its input either by making it concrete when the program requires (and at other times, judiciously chosen) or by collecting constraints along each trace. The conjunction of constraints for a trace is finally checked for feasibility by a specialized SAT solver that also generates real inputs as examples. Two of the example programs included in this evaluation, Mutt's UTF8 routine and Pintos' `printf` are there for comparison with EGT. The final coverage COMET achieves is comparable to that of EGT for these programs. DART and CUTE are similar except that they start with random testing, collecting constraint sets along traces and finally solving them to come up with new inputs that will generate traces corresponding to branches not yet taken. Although the approach is different from ours, there are similarities. They collect constraints and then solve to find inputs that expose new code, whereas we make use of a dynamic taint trace to suggest small modifications to input bytes that will expose new code. Perhaps the most fundamental difference between these combined static-dynamic approaches and our fully dynamic one is the ability to make use of a regression test suite. COMET finds small perturbations of inputs that expose new code, given a set of inputs. If the initial input is a single random one, then it will have difficulty exposing code that requires long and highly structured input. We imagine the same could be said of CUTE, DART, and EXE, that they would, for instance, be slow to produce traces for inputs that correspond to a well-formed *and interestingly long* pdf file. COMET can make immediate use of a test suite of pdf files to address this difficulty, collecting them along with a number of random inputs to assemble an initial corpus that leverages both well-formed and ill-formed inputs, to explore the code both in depth and in breadth, and to do so faster. Conversely, it is not apparent how the concrete/symbolic execution systems would make use of the sample traces a test suite would provide.

Dynamic taint propagation approaches have existed at least since Perl's `taintperl`, which was introduced in version 4.0 in 1991 [32]. This interpreter labeled all inputs, propagated that label to values computed from those inputs, and dynamically prevented `system`, `eval`, and other calls that took those values as arguments. Other modern scripting languages such as Ruby have taint modes that work similarly; however, all propagate taint at the variable rather than the byte level and have no way of differentiating between sources [31]. Recently, even PHP has been given the ability to propagate taint labels, and at the byte level [24]. Approaches for C began with Taintbochs and Taintcheck, both of which work at the byte level by maintaining a map of the taintedness of each byte in memory [5, 23, 34, 35]. Taintbochs also arranges to follow taint between programs and

into the kernel. However, both are slow as they work within Bochs and Valgrind [1, 22]. Other, more recent, research has implemented much faster taintedness maps for C and added some limited form of IIF in order to reduce false negatives [34, 35]. The chief distinction between all of these dynamic taint systems and COMET is that none provide the ability to query variables and trace taint back to specific input bytes. This ability is central to our approach, reducing the search over the input space dramatically.

6. CONCLUDING REMARKS

We have presented COMET, a new automated system for generating a test set for a program that maximizes line coverage. COMET outperforms random testing by a substantial margin on a set of 13 example programs of a few hundred lines each, uncovering 23% more lines for these examples, on average, and requiring less than twice as much time to do so. System tuning experiments indicate that similar gains might be had for as little as one-third the runtime of random testing. Additionally, our final corpus of test cases is six orders of magnitude smaller than that required by random testing, a dramatic reduction. Our immediate plans involve applying COMET to the name server `bind`, which we have begun. Initial experiments indicate that a hybrid approach, employing random testing selectively in the directed search will result in a system that is faster and uncovers even more code than COMET. Additionally, we believe it will be worthwhile to introduce some IIF (Implicit Information Flow [28]) into the taint graph in order to address the potential of false negatives.

REFERENCES

[1] Bochs: The cross platform ia-32 emulator. `https://bochs.sourceforge.net`.

[2] Thomas Ball. A theory of predicate-complete test coverage and generation. In *Proceedings of the Symposium on Formal Methods for Components and Objects*, 2004.

[3] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. Technical Report CSTR-2005-04 3, Stanford, 2005.

[4] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[5] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, 2004.

[6] Christoph Csallner and Yannis Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, July 18–20, 2006.

[7] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[8] M. Fisher II, D. Jin, G. Rothermel, and M. Burnett. Test reuse in the spreadsheet paradigm. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, November 2002.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI '05: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM Press, 2005.

[10] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using an iterative relaxation method. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1998.

[11] Bhargav Gulavani, Thomas Henzinger, Yamini Kannan, Aditya Nori, and Sriram Rajamani. Synergy: A new algorithm for property checking. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2006.

[12] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[13] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[14] Paul C. Jorgensen. *Software Testing, A Craftsman's Approach.* CRC Press, Boca Raton, FL, 2004.

[15] Rauli Kaksonen. A functional method for assessing protocol implementation security. Technical Report 448, VTT Electronics, 2001.

[16] Gregory M. Kapfhammer. *Software Testing*, chapter 105. CRC Press, Boca Rato, FL, second edition, 2004.

[17] Bogdan Korel. Automated test data generation for programs with procedures. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1996.

[18] Christopher C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, 2001.

[19] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of MacOS applications using random testing. In *Proceedings of the 1st International Workshop on Random Testing*, 2006.

[20] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conference on Compilier Construction*, 2002.

[21] John Nelder and Robert Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[22] Nicholas Nethercote and Julian Seward. A program supervision framework. In *Workshop on Runtime Verification*, 2003.

[23] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[24] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *International Information Security Conference*, 2005.

[25] William Press, Saul Teukolsky, William Vetterling, and Biran Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, UK, 1997.

[26] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, 1999.

[27] Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS 2004)*, 2004.

[28] Andrei Sabelfeld and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, 21(1):5–19, 2003.

[29] Securiteam. Mutt controlled imap server buffer overflow. `http://www.securiteam.com/unixfocus/5FP0T0U9FU.html+`.

[30] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2005.

[31] Dave Thomas. *Programming Ruby.* The Pragmatic Bookshelf, Raleigh, NC, 2005.

[32] Larry Wall, Tom Christiansen, and Randal Schwartz. *Programming Perl.* O'Reilly Media, Sebastapol, CA, 1996.

[33] David Wheeler. More than a gigabuck: Estimating GNU/Linux's size. `http://www.dwheeler.com/sloc`, 2001.

[34] Wei Xu, Sandeep Bhatkar, and R Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical report, State University of New York, Stony Brook, May 2004.

[35] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, 2006.

[36] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: Better together! In *International Symposium on Software Testing and Analysis*, 2006.

[37] Misha Zitser. Securing software: An evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology, 2004.

[38] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 97–106, 2004.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED *(From - To)*
28 March 2007	Technical	

4. TITLE AND SUBTITLE

Coverage Maximization Using Dynamic Taint Tracing

5a. CONTRACT NUMBER
FA8721-05-C-0002

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
10060

6. AUTHOR(S)

T.R. Leek, G.Z. Baker, R.E. Brown, M.A. Zhivich, and R.P. Lippmann

5d. PROJECT NUMBER

5e. TASK NUMBER
271

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AND ADDRESS(ES)

MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108

8. PERFORMING ORGANIZATION REPORT NUMBER
TR-1112

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Department of Homeland Security
245 Murray Lane, SW
Washington, D.C. 20528

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
ESC-TR-2006-079

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We present COMET, a system that automatically assembles a test suite for a C program to improve line coverage, and give initial results for a prototype implementation. COMET works dynamically, running the program under a variety of instrumentations in a feedback loop that adds new inputs to an initial corpus with each iteration. One instrumentation in particular is crucial to the success of this approach: dynamic taint tracing. Inputs are labeled as tainted at the byte level and all read/write pairs in the program are augmented to track the flow of taint between memory objects. This allows COMET to determine from which bytes of which inputs the variables in conditions derive, thereby dramatically narrowing the search over inputs necessary to expose new code. On a test set of 13 example programs, COMET improves upon the level of coverage reached in random testing by an average of 23% relative, takes only about twice the time, and requires a tiny fraction of the number of inputs to do so.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE	Same as report	41	19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified			